

Multiclass Software Defect Prediction Using an Optimized LLM CodeSentinel-BERT Based Framework

Vijayarajan S.M.¹, Ponnalagar V.², Ravikrishnan R.³,
Sarguna Gokul R.⁴

¹Associate Professor, ^{2,3,4}Student, Department of Computer Science and Engineering, NPR College of Engineering and Technology, Natham, Dindigul, India.

E-mail: ²sponnalagar58@gmail.com, ³raviramakrishnan430@gmail.com, ⁴sargunagokul@gmail.com

Abstract

The prediction of software defects is important to detect defects in source codes prior to their deployment but traditional techniques use selected features and limited binary classification. This research work proposes DefectGuard, a transformer-based multiclass software defect prediction system for C/C++ source code that utilizes CodeBERT fine-tuning. This custom dataset consists of 2,000 samples of manually collected and labeled C/C++ source codes and includes nine classes of defects, namely syntax, semantic, logic, linker, and runtime-related defects. The proposed pipeline includes preprocessing, stratified splitting, sub-word tokenization, CodeBERT representation learning, and multiclass classification. The experimental results show the accuracy, precision, recall, and F1-score of 98.8%, 96.4%, 95.8%, and 96.1%, respectively, on the test set. Additionally, ROC-AUC and confusion-matrix metrics are used to evaluate the class-level discrimination. Moreover, the proposed model is implemented in a real-time Flask-based web application, which can classify the source code into defect labels with confidence scores and language detection to provide the interpretation feedback.

Keywords: Software Defect Prediction, Source Code Analysis, Transformer Models, CodeBERT, Multiclass Classification, ROC-AUC Analysis, Real-Time Defect Detection.

1. Introduction

Software Defect Prediction (SDP) enables the detection of defective software at an early stage to minimize the effects of un-detected bugs during the development and maintenance process. The economic consequences of sub-optimal software testing were discussed by the researchers in [11], whereas Menzies et al. [4] used code static features to perform the task of defect prediction but the existing SDP systems mostly rely on hand-engineered software metrics and use defective vs. non-defective formulation to predict defects that limit their capability of capturing the semantics of source code along with identifying the nature of the defect.

To address the limitation of coarse-grained defect classification, this research proposes DefectGuard, which is a multiclass software defect predictor for C/C++ source code using CodeBERT as its base model. Here, nine classes have been considered, they are CLEAN, SIGFPE, SIGSEGV, SIGABRT, NZEC, LINKER, SYNTAX, LOGICAL, and SEMANTIC. Fine-tuning of the CodeBERT model is performed, and its performance has been compared with that of RoBERTa and GPT-2 models. Finally, the trained model is deployed as a Flask app for prediction and confidence scoring.

1.1 Contributions

This research presents DefectGuard, a CodeBERT-based system that classifies C/C++ source code into one of the nine software defect categories; performs a comparison between CodeBERT, RoBERTa, and GPT-2 on their performance for multiclass software defect prediction; and embeds the trained model into a Flask-based web application to classify defects in real-time, predict confidence score, and detect C/C++ language.

2. Related Works

The process of defect prediction using software defect prediction (SDP) has been using static software metrics and conventional machine learning algorithms. Menzies et al. [4] showed the utility of static code features in predicting defects in software while Lessmann et al. [5] compared different classification algorithms for SDP. Inadequate software testing, which has significant economic consequences, as stated by National Institute of Standards and Technology, Tassey. [11], further underlines the need for defect detection and quality control. However, these methods have laid important groundwork for SDP but heavily rely on hand-crafted features with

poor semantic representation of the source code.

Deep learning was used to enable the automatic learning of representations of code. White et al. [9] showed that neural architectures are capable of learning meaningful representations of code snippets, whereas Alon et al. [10] proposed structural representations of code, which are based on paths in an abstract syntax tree. Zou et al. [6] were able to show the power of deep learning in multiclass vulnerability detection. This indicates that the learned representations can learn features of source code that go beyond traditional software metrics.

Contextual representation learning was enhanced later with transformer architectures. BERT was introduced for bidirectional context representation learning by Devlin et al. [12], while RoBERTa was proposed by Liu et al. [2], as an improvement on BERT's pre-training process. Large scale autoregressive context representation learning is shown by GPT-2 by Radford et al. [3]. For the source-code understanding task, Feng et al. [1] came up with CodeBERT with code and natural language representation based on CodeSearchNet corpus by Husain et al. [8]. Guo et al. [7] enhanced code representation learning with GraphCodeBERT.

Overall, the development in the literature moves from defect prediction based on metrics to learning code representations and using code transformers. However, none of the aforementioned researched directly deal with the problem of nine-class classification of C/C++ source code snippets pertaining to different types of software defects. This sets the premise for the application of CodeBERT in multiclass software defect prediction.

3. Proposed Methodology

3.1 System Architecture

The proposed DefectGuard framework utilizes the sequential process for multi-class C/C++ software defects prediction that is shown in Figure 1. The source-code data goes through preprocessing and tokenization stages first and then is used as input for the CodeBERT classifier. CodeBERT creates context embeddings for programming language tokens and is trained for the task of nine-class software defects prediction [1]. The RoBERTa and GPT-2 models are used for the comparison. The results obtained by the model are transformed to class probabilities and are used to obtain the predicted defect class and confidence level, which is displayed via the DefectGuard web interface.

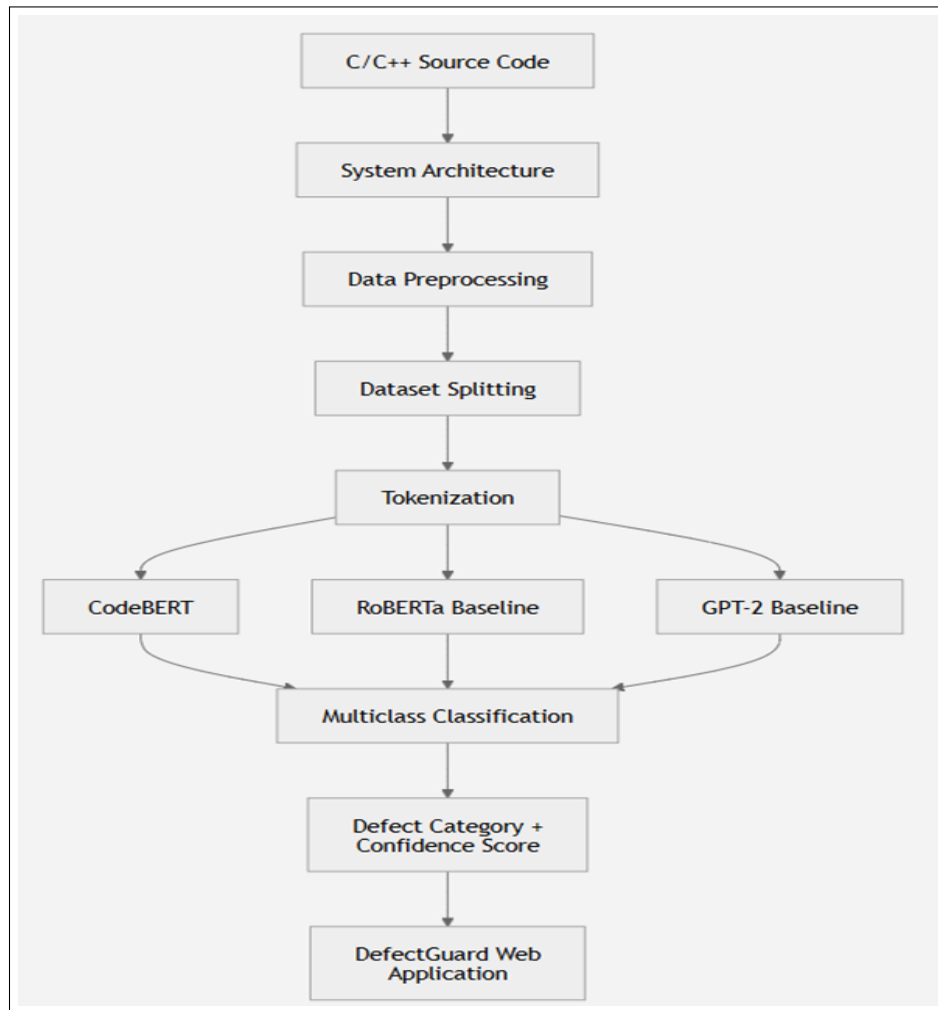


Figure 1. Workflow of the proposed DefectGuard-based multiclass software defect prediction system

3.2 Data Preprocessing Pipeline

The dataset used for the current research was created manually and sorted in `software_defect.csv` and comprises 2,000 C/C++ source code samples that are categorized into nine classes of software defects namely, CLEAN, SIGFPE, SIGSEGV, SIGABRT, NZEC, LINKER, SYNTAX, LOGICAL, and SEMANTIC. The source code samples have been manually compiled and sorted into their respective classes. The sample collection and categorization into nine classes is illustrated in Table 1. Before training the models, duplicates and incomplete records were eliminated, and leading and trailing white space normalization was completed, along with class label standardization and coding to numbers.

Table 1. Distribution of C/C++ source-code samples across defect categories

Defect Category	Description	Number of Samples
CLEAN	Correct source code without identified defects	208
SIGFPE	Floating-point exception	225
SIGSEGV	Segmentation fault	205
SIGABRT	Abnormal termination	223
NZEC	Non-zero exit code	226
LINKER	Linking-related error	217
SYNTAX	Syntax error	230
LOGICAL	Logical error	222
SEMANTIC	Semantic error	244
Total		2,000

Sample Data

Representative samples from the dataset are shown below:

```

• if (result == 5 { std::cout << "hi"; }
  → SYNTAX
• int main() { exit(208); } → NZEC
• #include <stdlib.h> int main() {
  abort(); } → SIGABRT
• while(true) { if(index > 0) break; index--; } → LOGICAL
• int result = 419; result.push back(5);
  → SEMANTIC

```

3.3 Dataset Splitting

After data preprocessing stage, the dataset is partitioned into training, validation, and test sets with stratified 60:20:20 partitioning. Stratification is employed to ensure that the proportion

of all the nine defect classes is maintained within the three datasets. The training dataset is employed for fine-tuning the model, while the validation dataset is utilized for selecting the best model and observing the change in its hyperparameters. On the other hand, the test set is reserved for testing and ensuring there is no impact on the training process of the model.

3.4 Tokenization

The preprocessed C/C++ code snippets are then represented by their token sequences through the RoBERTa Byte-Pair Encoding (BPE) tokenizer that is part of the CodeBERT model. A fixed length of 128 tokens is set as the maximum for each token sequence, whereby sequences that exceed this limit will be truncated to fit this length while sequences that fall below this length will be padded. This tokenization step ensures that programming language tokens are preserved, but the source code is then represented numerically.

3.5 Model: CodeBERT

The selected CodeBERT as the main model is due to its pre-training on programming language and natural language data which allows context-based token representation of the source code. The CodeBERT model was introduced by Feng et al. [1] as the bimodal pre-trained model intended for the programming language understanding. In the proposed system, the pre-trained CodeBERT model is fine-tuned on the preprocessed C/C++ training samples for the nine-class defect prediction problem. The representation of the source code is fed into the classification layer to predict logits associated with nine classes of defects.

3.6 Baseline Models

RoBERTa and GPT-2 are two transformer models that have been pretrained and serve as baseline models for comparing the proposed CodeBERT classifier. RoBERTa was presented by Liu et al. [2] as a refined version of BERT pretraining, and GPT-2 was presented by Radford et al. [3] as an autoregressive transformer language model. Both of them are fine-tuned on the same C/C++ dataset in the same nine-class prediction task, thus providing the same conditions of comparison for CodeBERT. The classification results from both of them are analyzed under the same experimental environment.

3.7 Inference and Classification

During inference, the preprocessed and tokenized C/C++ source code is fed into the trained transformer network model, from which class logits for all nine defect classes are

obtained. Softmax function converts logits to probabilities, and the class having the highest probability is selected as the predicted defect class. The probability of that class is considered the prediction confidence. Finally, the output of inference is presented via the DefectGuard interface.

3.8 Loss Function and Optimizer

To optimize the model parameters, the categorical cross-entropy loss function is utilized in order to reduce the error between the prediction probabilities and the true defect class. For N training samples and $C = 9$ defect categories, the loss is defined as:

$$\mathcal{L}_{CE} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c}) \quad (1)$$

where $y_{i,c}$ represents the true class indicator and $\hat{y}_{i,c}$ denotes the predicted probability for class c . AdamW optimizer is used to update the parameters of the model during the fine-tuning process. The parameter update can be defined as:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} + \lambda \theta_t \right) \quad (2)$$

where θ_t denotes the model parameters, η is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moment estimates, ϵ is a numerical stability term, and λ represents the weight-decay coefficient. Through the process of optimization, it becomes possible to fine-tune the pretrained CodeBERT parameters for predicting defects in nine classes.

3.9 Experimental Setup

The experimental configuration follows the specifications described in the literature to ensure consistency in the experimental evaluation of the CodeBERT, RoBERTa, and GPT-2 models. The major parameters for the training process of all three models such as the number of epochs, learning rate, batch size, sequence length, optimization strategy, and computational setup are presented in Table 2.

Table 2. Experimental configuration of the defect prediction models

Parameter	Configuration
Dataset	software_defect.csv
Programming Language	C/C++
Number of Classes	9
Train–Validation–Test Split	60:20:20
Maximum Sequence Length	128
Batch Size	8
Learning Rate	2×10^{-5}
Number of Epochs	5
Optimizer	AdamW
Weight Decay	0.01
Primary Model	CodeBERT
Baseline Models	RoBERTa, GPT-2
Evaluation Metrics	Accuracy, Precision, Recall, F1-Score, ROC-AUC
Configuration	NVIDIA GPU-based training environment

4. Results and Discussion

4.1 Performance of Proposed Model

The proposed CodeBERT model demonstrates strong classification performance on the test set, obtaining 98.8% accuracy, 96.4% precision, 95.8% recall, and 96.1% F1-score. From the high accuracy rate, it is evident that the classification results of the presented model are very good in assigning most of the C/C++ source-code samples to their corresponding nine defect categories. The obtained precision rate indicates that the predictions made by the model regarding the defect categories are very accurate. The recall rate depicts how well the model

is able to identify most of the defect instances present in the test data. The F1-score gives an indication of the classification capabilities of the presented model based on precision and recall rates. The complete performance of the proposed model is summarized in Table 3.

Table 3. Performance of the proposed CodeBERT model

Model	CodeBERT
Accuracy (%)	98.8
Precision (%)	96.4
Recall (%)	95.8
F1-Score (%)	96.1

4.2 Evaluation Metrics

The proposed CodeBERT model is tested through One-vs-Rest ROC-AUC analysis to determine its discriminative performance in relation to the nine types of defects. In the figure 2, the AUC score for each of the nine categories is 1.00. It therefore means that all nine ROC curves overlap closely, presenting one distinct curve.

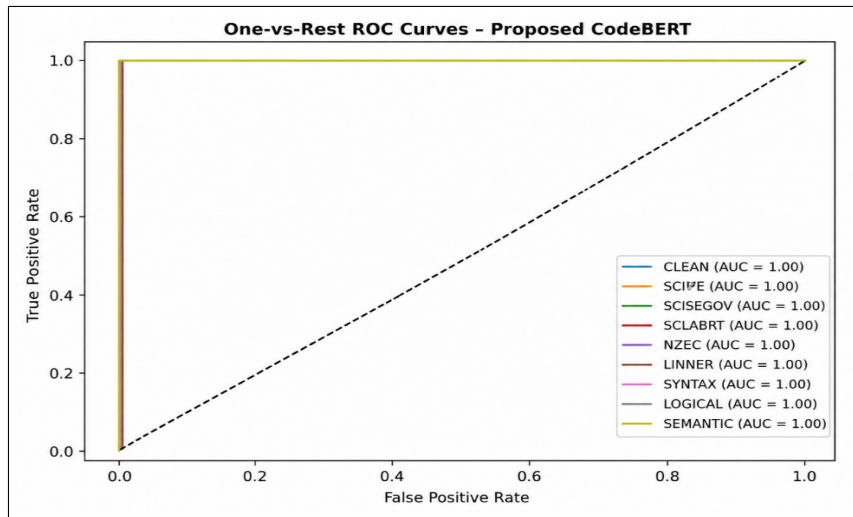


Figure 2. ROC-AUC analysis of CodeBERT

Figure 2 contains a ROC analysis that is a class-based evaluation method to supplement the aggregate results discussed in Table 3. The overlapping curves among the nine classes demonstrate that the proposed model exhibits the same level of discrimination in all classes of defects and not just some weak classes.

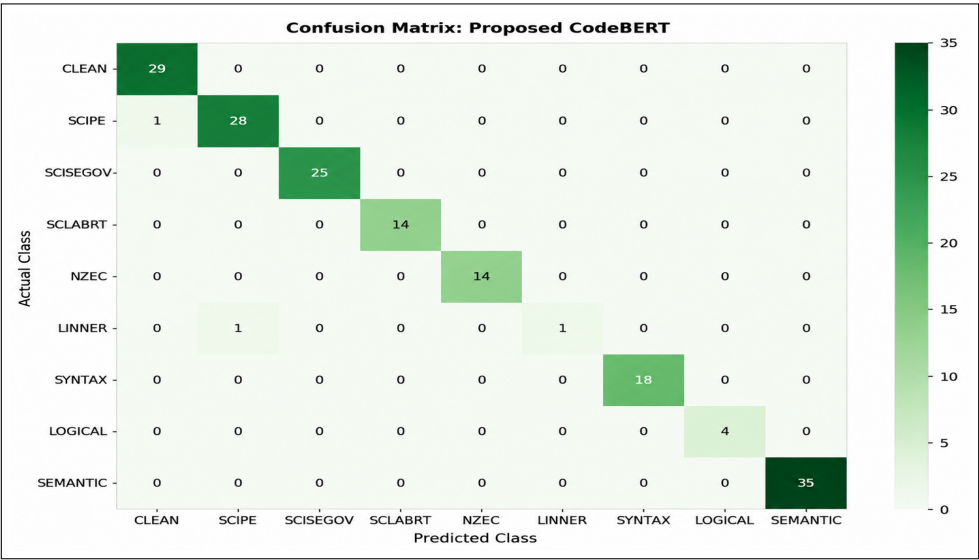


Figure 3. Class-wise confusion matrix of CodeBERT

Moreover, the class-level prediction performance is analyzed through the 9×9 confusion matrix given in Figure 3. The values in the main diagonal are those instances that have been correctly classified, while the values that do not lie in the main diagonal are instances which have been incorrectly classified between different classes. This is another indication of the reliable multiclass classification performance of the model.

4.3 Real-Time Prediction Interface

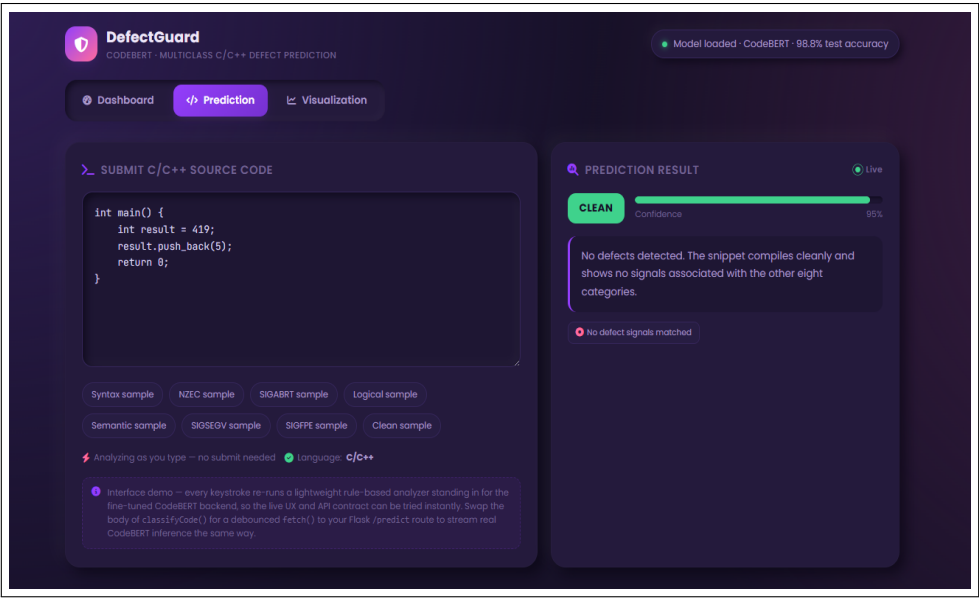


Figure 4. Real-time C/C++ defect prediction interface of DefectGuard

The proposed interface, DefectGuard offers a way of submitting C/C++ source code and receiving instant defects predictions. As seen from Figure 4, there is a prediction page

that comprises such elements as source code submission area, samples of defects categories, language detection and prediction area. The result of the classification process is represented on the interface along with confidence values.

The prediction panel also provides a brief description related to the predicted class and tells about the detection of any defect signal. In this particular example, the inputted C/C++ code snippet is predicted to be CLEAN, and its confidence value is equal to 95%, whereas the interface says that there are no defect signals found. Thus, the way of presenting the prediction results is illustrated. Furthermore, the interface incorporates two parts – the status of the model and the live prediction status. The language indicator shows the submitted language as C/C++, while the live prediction status gives an instant feedback on the analysis process.

5. Conclusion and Future Work

DefectGuard is an advanced multiclass prediction tool used for C/C++ source codes by utilizing CodeBERT. This solution is an example of combining code preprocessing, contextual feature learning, defect prediction and confidence-based decision making into one unified predictive pipeline. In contrast to the traditional binary defect prediction method, the nine classes allow to predict particular types of software defect, and a live demonstration Flask app shows to this solution can be incorporated into practice by providing an example of training and using the model in real time. Moreover, DefectGuard allows to automate defect prediction by providing code classification and language detection, confidence level, and prediction feedback. Further research includes extension of the dataset to other programming languages, exploration of the data flow-aware approaches like GraphCodeBERT, and multilabel defect prediction.

References

- [1] Feng, Zhangyin, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou et al. “Codebert: A Pre-trained Model for Programming and Natural Languages.” In Findings of the Association for Computational Linguistics: EMNLP, 2020, 1536-1547.
- [2] Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. “Roberta: A Robustly Optimized Bert Pretraining Approach.” arXiv preprint, 2019, 1-13.
- [3] Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever.

- “Language Models are Unsupervised Multitask Learners.” OpenAI Blog, 2019, vol. 1, no. 8: 9.
- [4] Menzies, Tim, Jeremy Greenwald, and Art Frank. “Data Mining Static Code Attributes to Learn Defect Predictors.” *IEEE Transactions on Software Engineering*, 2007, vol. 33, no. 1: 2-13.
- [5] Lessmann, Stefan, Bart Baesens, Christophe Mues, and Swantje Pietsch. “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.” *IEEE Transactions on Software Engineering*, 2008, vol. 34, no. 4: 485-496.
- [6] Zou, Deqing, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. “ μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection.” *IEEE Transactions on Dependable and Secure Computing*, 2019, vol. 18, no. 5: 2224-2236.
- [7] Guo, Daya, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou et al. “Graphcodebert: Pre-training Code Representations with Data Flow.” *arXiv preprint*, 2020.
- [8] Husain, Hamel, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. “Codesearchnet Challenge: Evaluating the State of Semantic Code Search.” *arXiv preprint*, 2019.
- [9] White, Martin, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. “Deep Learning Code Fragments for Code Clone Detection.” In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, 87-98.
- [10] Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav. “code2vec: Learning Distributed Representations of Code.” *Proceedings of the ACM on Programming Languages*, 2019, vol. 3, no. POPL: 1-29.
- [11] Gregory Tasse. “The Economic Impacts of Inadequate Infrastructure for Software Testing.” *National Institute of Standards and Technology*, 2002, vol. 1, no. 2002: 1.
- [12] Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding.” In *Proceedings of the 2019 Conference of The North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, vol. 1: 4171-4186.